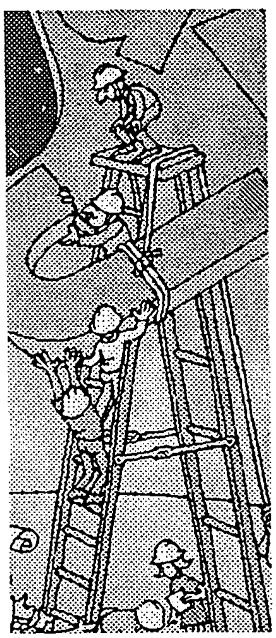
JAVA-CENTRIC DISTRIBUTED COMPUTING

Ann Wollrath

Jim Waldo

Roger Riggs

JavaSoft



The Java language has changed the assumptions governing distributed computing systems.

One of the most common ways of communicating between processes in a distributed system is the remote procedure call.[1] This communication device allows one process to call a function that runs in some other process, perhaps on a different machine. In the traditional single-process programming model, control moves from one point in a program to another point in the same program. With RPC, however, control can flow from a point in one program to a point in another program on another machine.

The usual assumptions about RPC systems reflect the reality of most distributed environments. We generally assume, for example, that a machine making an RPC can know nothing about the architecture or the instruction set of the machine it is calling. Further, we generally assume (although not always, see the adjacent box) that the programming language used to implement the called program is also unknown. Most distributed systems contain a variety of hardware, operating systems, and services—each implemented in the most convenient language—so these assumptions make sense.

The Java language and platform[2,3] provide a base for distributed computing that changes several of these assumptions. In particular, the Java Virtual Machine allows a group of Java-enabled machines to be treated as a homogeneous group rather than a heterogeneous group—despite possible differences in the machine architectures and underlying operating systems. Java also makes it possible to safely and dynamically load code in a running Java process. Together, these features allow a system to invoke methods on remote objects, which can move code associated with language-level objects from the calling process to the process called and vice versa. Combining these qualities with a language-centric design not only significantly simplifies traditional RPC systems, it adds functionality that was previously not possible.

We designed Java Remote Method Invocation (RMI) to support pure-Java distributed objects in a seamless manner. RMI allows calls to be made between Java objects in different virtual machines, even on different physical machines. RMI's language-centric design allows distributed programming to take place entirely in Java, making available such language features as the Java type system and garbage collection. Coupled with the Java platform's code portability, RMI's language specificity greatly simplifies distributed object programming. In addition, the pure-Java approach enables RMI to support distributed polymorphism and pass-by-value objects in remote calls, features which traditional distributed object systems cannot support.

RMI's structure

The RMI system consists of several basic layers; a specific interface and protocol define the boundary at each layer. Each layer is independent of the next and can be replaced by an alternate implementation without affecting the other layers. For example, the current transport implementation in the Java Development Kit 1.1 is based on TCP (Transmission Control Protocol) using Java sockets. However, a transport based on UDP (User Datagram Protocol) could be used as well.

To accomplish transparent transmission of objects from one address space to another, the RMI system uses Java Object Serialization. Another technique, which we call dynamic stub loading, supports client-side stubs that implement the same set of remote interfaces as a remote object itself. Because a stub supporting the exact set of remote types is available to the client of a remote object, that client can use Java's built-in oper-

0272-1732/97/$10.00 © 1997 IEEE

Distributed computing with RPC

Traditional approaches to distributed computing with remote procedure call have made very constraining assumptions about the environment in which those systems must run. In particular, we have usually assumed that a system would run on a variety of hardware. This hardware would connect processes running code written in a variety of languages, compiled to the machine code for the various platforms.

The key to allowing communication within these constraints in most RPC systems is the notion of an interface, which describes a set of entry points that can be called from a remote machine. We describe these interfaces using an interface definition language (IDL),[1] which describes both the entry points and the data types passed (either as parameters or as return values) between the communicating processes.

Once we have described an interface in an IDL, we can run a compiler over the interface description to generate the code needed for the communication. The calling process—sometimes known as the client—includes code often known as a stub. This stub code acts as a local surrogate for the remote process servicing the call. It takes any parameters passed to the call, packages them so they can be transported across the network, and adds an identification of the entry point to be called.

The client then transmits this package over the network; it is received by the other piece of code generated by the IDL compiler, known as the skeleton. The skeleton's purpose is to take the incoming network byte stream and reconstruct the parameters, identify the entry point to be called, and make the up call to the entry point with those parameters. Once the call processing is complete, the skeleton converts any return values into a form that can be sent over the network and sends this back to the original stub code.

The stub, in turn, converts the network representation into something the local caller can understand and passes the return values back to that caller. Since a compiler generates stub and skeleton code for a particular machine and implementation language, programmers need not worry about the translation to or from the network representation. Nor need they worry about the correctness of the translation from that representation to the client's or server's particular environment.

IDL design reflects some of the best programming practices in use when those languages were invented. The early IDLs, written when object-oriented languages were just coming into use, are excellent examples of data abstraction languages. Distribution requires a strict distinction between the interface (what is to be done by a server) and the implementation (how that thing is done). This is reflected in the IDLs' ability to declare the interface to a service and then implement that interface in any way the programmer deems necessary.

More recent IDLs, like that found in CORBA,[2] add a notion of inheritance to the language, becoming more object-based. In these languages, we can define an interface that supports an extension of a more basic interface, yet a server implementing the extended interface can still be considered to implement that basic interface. This gives the server a notion of polymorphism, in that an implementation of a server supporting the more derived interface is also treated as an implementation of a server that supports the base interface. This has the effect of treating the server as though it were of both the base type and the derived type within the distributed system.

ators for casting and type checking remote objects. Traditional, language-neutral distributed object systems cannot use built-in operators for casting and type checking, because they do not integrate distributed objects into any language's type system.

Architectural overview

The RMI system's layers consist of the following:

- *stub/skeleton*—client-side stubs (proxies) and server-side skeletons (dispatchers);
- *remote reference*—reference and invocation behavior (for example, unicast and multicast);
- *transport*—connection setup and management, remote object tracking; and
- *distributed garbage collection*—reference-counting garbage collection of remote objects.

RMI is a layer on top of the Java Virtual Machine, so it leverages the Java system's built-in garbage collection, security, and class-loading mechanisms. The application layer sits on top of the RMI system.

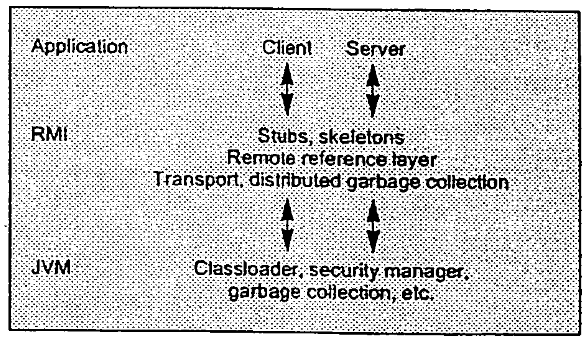


Figure 1. RMI system architecture.

Figure 1 shows the RMI system's layers. A remote method invocation from a client to a remote server object travels down through the RMI system's layers to the client-side transport layer. Next, the client sends the invocation (potentially via a network communication) to the server-side transport. There, the invocation travels up through the server-side trans-

Distributed computing with RPC (continued)

Systems such as CORBA and DCOM still pay a price for their assumption of an environment containing heterogeneous computing platforms and multiple languages. The range of values that can be passed either as arguments or return values is limited to those that can be represented in all of the implementation languages supported. Also, the object orientation is limited to objects that are passed by reference (because they are remote objects) or objects that are predefined as part of the overall distributed system framework. There is also a conceptual price: programmers using these systems must deal with (at least) two languages—one for the interface definitions and one for the implementation—and must be able to map between the two.

The other price for heterogeneity in such systems is the requirement that types passed and returned be exactly what was declared in the interface. The sending and receiving processes may be written in any language and compiled into any instruction set. Thus, there is no way to extend the range of permissible values transmitted on the fly and still ensure that the value will be properly interpreted upon receipt.

Some research groups have taken a different approach, exemplified by the Network Objects system developed as part of the Modula-3 environment.[3] This system provides RPC-like functionality, but in a language-dependent fashion. Rather than having a separate IDL, the system declares remote interfaces in the same way as other interfaces in the language. This simplifies programming, as there is no need to map from an IDL to an implementation language; the two are the same.

An additional advantage of the language-centric approach is that language-level objects can pass from one process to another. This is possible, in part, because the implementation language is the same for both communicating parties; this allows the use of a single type system. Thus, there is some guarantee that the same type will appear in both processes, which allows the parameters and return values to be sent over the wire and reconstructed.

The only complication to the ability to send objects from one process to another is the polymorphic nature of the type system in Modula-3. A Modula-3 object can have many types, since one type can be defined as an extension of another type. Any object that is of the extended type is also of the base type.

In the Modula-3 Network Objects system, the distributed type matching is complicated by the fact that it is always possible for a remote call to pass a derived type. There is no guarantee that the code corresponding to this derived type is present in the receiving process. Unless the code for the type that is passed from one process to another is present in both processes, serious errors can occur. The Network Object system mitigates this problem by using the closest matching type for a reference to a network object. However, pass-by-value objects cannot perform this kind of matching.

References

1. M. Kong et al., *Network Computing System Reference Manual*, Prentice Hall, Englewood Cliffs, N.J., 1987.
2. "Common Object Request Broker: Architecture and Specification," OMG Document No. 91.12.1, The Object Management Group, Framingham, Mass., 1991.
3. A.D. Birrell et al., "Network Objects," Tech. Report 115, Digital Equipment Corporation Systems Research Center, Palo Alto, Calif. 1994.

port layer to the server. The rest of this section summarizes the functionality at each of RMI's layers.

A client invoking a method on a remote server object actually uses a stub (proxy) for the remote object as a conduit to that object. A client-held reference to a remote object, therefore, is a reference to a local stub. This stub is an implementation of the remote object's remote interfaces; it forwards invocation requests to that server object using the remote reference layer.

This layer carries out the semantics of the type of invocation. Each remote object implementation chooses its own invocation semantics. For example, the semantics can indicate that communication to the server is unicast (point-to-point), or that the server is part of a multicast group (to accomplish server replication). Thus, the RMI architecture provides a framework for adding new types of remote object communication in the future.

The remote reference layer also handles reference semantics for the server. For example, the remote reference layer can handle live and/or persistent references to remote objects. A persistent reference allows objects that are not currently servicing calls to be saved to a permanent store; this frees resources used by those objects. The next call to such an object reactivates it, returning it to the state it was in prior to being saved to persistent store.

The transport layer is responsible for connection setup, connection management, and keeping track of and dispatching to remote objects (the targets of remote calls) in the transport's address space. To dispatch to a remote object, the transport forwards the remote call up to the remote reference layer (specific to the server). After performing any necessary server-side behavior, the remote reference layer hands off the request to the server-side skeleton. The skeleton makes an up call to the remote object implementation, which carries out the actual method call.

The return value of a call travels back through the skeleton, remote reference layer, and transport on the server side, and then up through the transport, remote reference layer, and stub on the client side. Figure 2 shows the anatomy of an RMI call. In the figure, the remote object (obj) is represented by the client-side stub for the remote object. This stub implements all the remote interfaces (and thus remote methods of those interfaces) that the actual remote implementation object supports.

A call to "method" on stub "obj" marshals the arguments for the call and sends the request to the host containing the remote object. The client-side transport initiates a connection to the remote virtual machine and sends a message encoding the RMI call.

The server-side transport receives the message, looks up the skeleton for the remote object, and dispatches the call to the skeleton. The skeleton unmarshals the call's arguments and makes the up call to the actual remote object implementation. When the implementation completes its method by returning a result or throwing an exception, the skeleton marshals the result. Then, the transport sends the return result to the originating virtual machine. Finally, the stub initiating the call unmarshals the return result and either returns the value or throws the exception.

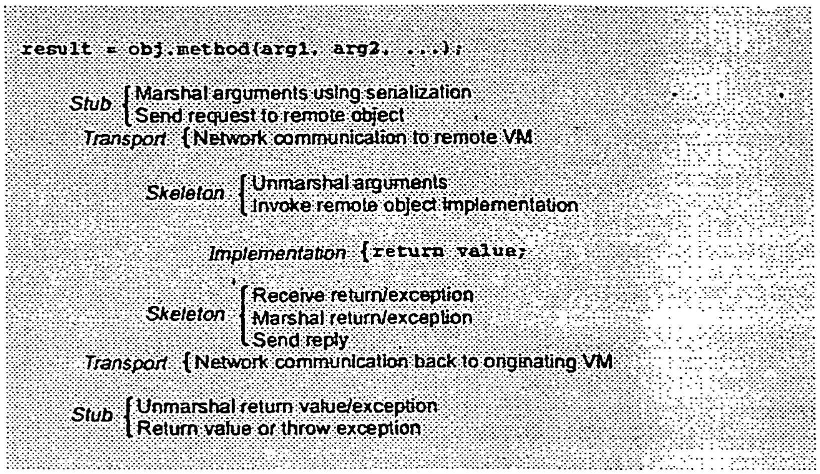


Figure 2. Anatomy of a call.

What is not depicted in Figure 2 is the interaction of the stub with the remote reference layer. The stub interacts directly with the remote reference layer, not with the transport. Any call the stub initiates goes through the reference layer to ensure appropriate reference semantics.

For example, if a remote reference included connection retry behavior, the remote reference would attempt a connection retry on behalf of the stub. When the connection was successful, the underlying transport for that reference type would take over, marshaling the arguments and sending the RMI call.

Another kind of remote reference is one that carries out multicast to a set of replicas for a remote object. (RMI does not yet provide this behavior, but it will in the future.) The stub for the remote object calls through the remote reference layer, which in turn communicates to the replica set via the transport. The reference layer provides the abstraction needed, so that multiple reference semantics can be carried out without a change to the stub's form. A stub simply instructs the layer beneath to initiate a call, marshal and unmarshal parameters, and return results. Various behaviors can take place beneath the stub without necessitating a change to the stub itself.

Object serialization

The RMI system can pass parameters and return values either by reference or by value. If the object to be passed is itself a remote object, RMI passes a remote reference to that object. However, if the object is not a remote object, RMI passes a copy of the object to the receiver, thus giving the effect of pass-by-value.

RMI uses Java Object Serialization to marshal and unmarshal parameters and return values. Object serialization provides a simple, easy-to-use mechanism for accurately making copies of objects across time and space. It encodes objects from one virtual machine into a stream of bytes. The stream can be passed through a network to another virtual machine or saved in a file or database for later retrieval. Later or elsewhere, object deserialization can construct copies of the original objects from the stream.

Objects are likely to contain references to other objects, and the relationships between objects are integral to the meaning and function of each object. Object serialization traverses graphs of objects and maintains the relationships so that the graph topology can be recreated.

The default serialization mechanism makes it simple to save and restore objects. All that is needed is for the class to declare that it implements the java.lang.Serializable interface. This lets the system write or read the object, and does not require the class to implement any special methods to handle serialization. Only when the default behavior is not sufficient does the class implement methods to customize the fields and values written to and read from the stream. The class assumes control for writing and reading the stream's state by implementing methods for writes and reads. A typical use of this function would be to write a more compact representation of a sparse data structure.

ObjectOutputStream implements methods for writing objects to a stream. The writeObject method encodes the object using information in the virtual machine for the object's class and superclasses. Serialization iterates through each class from the highest serializable class down to the object's actual class. For each class, the method writes the nontransient and nonstatic public, protected, package, and private fields to the stream. For primitive types, it encodes the values in the stream. In addition, writeObject writes fields holding references to objects to the stream; these include arrays, strings, and classes. If the class has its own methods implemented for writeObject and readObject, these supersede the default mechanism.

ObjectInputStream implements the complementary readObject method to read objects from a stream. Reading an object consists of reading its class, allocating a new instance, and restoring the contents for each of the object's

Traditional, language-neutral distributed object systems truncate the parameter or return value to the declared type in the remote method signature. With RMI, this is not the case.

classes, from highest serializable class down to the actual class. The default deserialization reads values for primitive types from the stream and assigns them to fields. For object fields, a call to readObject reads the object from the stream. As each object is assigned to a field, object serialization checks that the object's type can be assigned to the field.

Object graphs may be written to the stream; these are reconstructed with the same topology. Each ObjectOutputStream keeps a reference to each object written. The first time an object is written to the stream it is assigned a handle, and the handle remains with the reference to the object. When a new object is written, it is compared with the references for previously written objects. If the object already appears in the stream, it is encoded using the handle. Thus, each object is written to the stream exactly once, so all references to an object within the stream refer to the same instance. This naturally limits the graph traversal and assigns a unique number to each object within the stream.

Essential to transporting the object accurately is identifying the object's actual type. The goal is to retain an object's semantics across serialization and deserialization. Classes are identified by name and signature, and the stream includes the class's supertypes as well. When objects are being read from the stream, the normal class-loading mechanisms retrieve the class and verify its signature. This signature is important confirmation that the same class has been found. When methods or fields are added to a class as it evolves, the signature can be declared so that the class-loading mechanisms can identify the class as compatible with previous versions.

The mechanisms for object serialization allow subclassing of the output and input streams for class annotation and object substitution. The overridable annotateClass method on the output stream is called for each class written to the stream. A subclass can insert additional information about the class into the stream. A corresponding resolveClass method is called as each class object is read from the stream; it reads the additional information and uses it to find the appropriate class.

Before each object is serialized for the first time, the stream's replaceObject method gives the stream a chance to examine and act on the object. A corresponding resolveObject method is called on the input stream after an object is read from the stream and before the object is returned. These callbacks to the stream allow suitable replacements of one object with another. The substituted objects are always type-checked before assignment, so substitution does not compromise the objects' integrity.

Passing by true type

Because RMI uses object serialization to pass parameters and return values in RMI calls, it can pass the true type of an object from one virtual machine to another in a remote call. Thus, a remote method may define a formal parameter or return value as a specific Java type, but the remote call may pass a subtype of the declared type that preserves the object's true type on the receiving side.

Traditional, language-neutral distributed object systems truncate the parameter or return value to the declared type in the remote method signature. With RMI, this is not the case. The ability to pass a complete object without changing its type is very important in object-oriented programs. RMI preserves the basic object-oriented notion of polymorphism. Any system that does not allow object polymorphism is not truly object-oriented, but simply object-based:

> In such [object-based] programs, types are generally viewed as static, whereas objects typically have a much more dynamic nature, which is somewhat constrained by the existence of static binding and monomorphism.[4]

The RMI system does not constrain objects' "dynamic nature."

Downloading code

Remote procedure call systems must generate client-side stub code and link it into a client before making an RPC. This code can be either statically linked to the client or dynamically linked at runtime via libraries available locally or over a network file system. With either static or dynamic linking, the specific code that handles an RPC must be available to the client machine in advance in compiled form.

RMI generalizes this technique, using the Java system's dynamic class-loading mechanism at runtime to load (in Java's architecture-neutral bytecode format) the classes that handle method invocations on a remote object. The system downloads the following classes during an RMI call:

- classes of remote objects and their interfaces;
- stub and skeleton classes (created by the rmic stub compiler) that serve as proxies for remote objects; and
- other classes used directly in an RMI-based application, such as parameters to, or return values from, remote method invocations.

In addition to class loaders, dynamic class loading employs two other mechanisms: the Java Object Serialization system to transmit class locations over the wire and a security manager to secure the classes loaded.

As stated earlier, RMI uses object streams to transmit parameters and return values for remote calls. RMI also places

additional information in the call stream to support dynamic code loading. Object serialization allows the RMI runtime to annotate the stream with class-specific information when a class descriptor is written. RMI implements ObjectOutputStream's annotateClass method to write the code base URL at which the class can be downloaded.

On the receiving end, RMI uses ObjectInputStream's resolveClass method to read the code base URL embedded in the stream. If the class is not already defined locally, resolveClass then downloads the class from that location. Thus, a class's location (a URL) travels in the stream along with the objects of that class.

```
public interface StockWatch extends java.rmi.Remote {

    /** Request notification of stock updates. */
    Stock watch(String stock, StockNotify obj)
        throws StockNotFoundException, RemoteException;

    /** Cancel request for stock updates for a particular stock. */
    void cancel(String stock, StockNotify obj) throws RemoteException;

    /** Returns an array of stock update information for the stocks
     * already registered by the remote object. */
    Stock[] list(StockNotify obj) throws RemoteException;

    /** Cancel all requests for stock updates for the remote object. */
    void cancelAll(StockNotify obj) throws RemoteException;
}

public interface StockNotify extends java.rmi.Remote {

    /** Notification of stock updates for a particular time. */
    void update (Date date, Stock[] stocks) throws RemoteException;
}
```

Figure 3. Stock notification service remote interfaces. These interfaces define those methods that can be called from another virtual machine.

A class loaded by RMI is subject to the security restrictions put in place by the java.lang.SecurityManager installed for the virtual machine downloading that class. Browsers define an applet security manager for applets. This security manager prevents applets from accessing files on disk or opening network connections apart from those to the applet's host of origin; it also institutes other restrictions. For classes to be downloaded into applets or applications as a result of remote calls, RMI requires a security manager to protect the application and host from potential harm.

Garbage collection and leasing

In a distributed system, just as in a local system, it is desirable to automatically delete remote objects that are no longer referenced by any client. Automatic garbage collection frees the programmer from keeping track of remote objects' clients to determine when the program can terminate safely. RMI uses a reference-counting garbage collection algorithm similar to the one Birrell describes.[5]

To accomplish reference-counting garbage collection, the RMI runtime keeps track of all live remote references within each Java Virtual Machine. When a live reference enters a virtual machine, the runtime increments its reference count. Upon receiving the first reference to a specific remote object, the client's RMI runtime sends a referenced (dirty) message to the remote object's RMI runtime. This way, a remote object's runtime keeps track of all clients that hold references to objects residing in its virtual machine.

When a given virtual machine discovers a live reference to be unreferenced, the finalization of that reference decrements the count. When the client virtual machine has discarded the last reference to a specific remote object, the runtime sends an unreferenced (clean) message to the server. Monotonically increasing sequence numbers maintain ordering among the dirty and clean calls. Interface java.rmi.dgc.DGC embodies the server side of RMI's distributed garbage collector.

To handle client failure, the client holding a reference to a remote object leases the reference for a period of time. The lease period starts when the remote object's runtime receives the dirty call. It is the client RMI runtime's responsibility to renew such leases before they expire; it does this by making additional dirty calls for the remote references it holds. If the client does not renew a lease before it expires, the distributed garbage collector assumes that the client no longer references that remote object.

RMI example

In this section, we'll walk through an example of using RMI to implement a stock notification service. The complete code for this example is available as part of the Java Developer's Kit 1.1. The example application consists of two basic remote objects:

- a stock server that accepts requests from clients that wish to be notified when specific stocks are updated, and
- an applet that participates both as a client of the stock service and as the entity notified when stocks are updated (thus, the applet is also a server).

The applet registers itself with the stock server for price updates of various stocks. The stock server keeps track of the remote objects requiring notification of stock updates, and sends periodic price updates to these remote objects (implemented as applets). The applet displays the stock prices in a graph that it updates dynamically as it receives stock information from the stock server.

To write an applet or application using RMI, we first need to define the remote interfaces to the remote objects in our application. A remote interface is a Java interface that declares all the methods that may be invoked by clients of

```
/**
 * Start up the stock server; also creates a registry so
 * that the StockApplet can lookup the server.
 */
public static void main(String args[])
{
	// Create and install the security manager
	System.setSecurityManager
			(new RMISecurityManager());

	try {
		LocateRegistry.createRegistry(2005);
		StockServer server = new StockServer();
		Naming.rebind
			("//:2005/example.stock.StockServer", server);
	} catch (Exception e) {
		System.out.println
			("StockServer.main: an exception occurred: " +
					e.getMessage());
		e.printStackTrace();
	}
  }
}
```

Figure 4. The main( ) method to create and install the stock service.

```
/** Request notification of stock updates. */
public synchronized Stock watch (String stock,
					StockNotify obj)
	throws StockNotFoundException
{
	System.out.println("StockServer.watch: " + stock );

	if (!stockTable.containsKey(stock))
		throw new StockNotFoundException(stock);

	Vector stocks = (Vector)notifyTable.get(obj);

	// register interested party...
	if (stocks == null) {
		stocks = new Vector();
		notifyTable.put(obj, stocks);
	}

	// add stock to list
	if (!stocks.contains(stock)) {
		stocks.addElement(stock);
	}

	// start thread to notify watchers...
	if (notifier == null) {
		notifier = new Thread(this, "StockNotifier");
		notifier.start();
	}
	return (Stock)stockTable.get(stock);
}
```

Figure 5. Implementation of a remote method defined in the StockWatch interface.

the remote object. Such clients make calls to remote interfaces, not to the implementation classes of those interfaces. RMI identifies remote interfaces as interfaces that extend the abstract interface java.rmi.Remote.

The two remote interfaces of interest in the stock service application are StockWatch and StockNotify. The StockWatch interface is the interface to the stock server; the applet implements the StockNotify interface to be notified of stock price updates. (See Figure 3.)

Remote objects in the RMI system must implement an interface that extends the java.rmi.Remote interface; both StockWatch and StockNotify do this. Any method in an interface that extends the java.rmi.Remote interface must declare that it may throw an exception of type RemoteException. This RMI-generated exception signals problems with communication between the client and server of the particular call.

Next, we implement the stock service as a simple remote object that supports unicast (point-to-point) reference semantics. The stock service directly implements the StockWatch remote interface. The simplest way to implement a server is to extend the java.rmi.server.UnicastRemoteObject class, which provides simple point-to-point reference semantics. During construction, a UnicastRemoteObject makes itself available to the RMI runtime to accept requests from clients—that is, the object is "exported to the RMI runtime" when it is created. The StockServer constructor must include RemoteException in its "throws" clause, because during construction the object may encounter an error if it cannot be exported to the RMI runtime.

```
public class StockServer
	extends UnicastRemoteObject
	implements StockWatch, Runnable
{
	/** Construct the stock server */
	public StockServer() throws
				RemoteException
	{
		super();
		/* initialize server data
				structures... */
	}
```

This segment shows the constructor for the StockServer; it also shows the beginning of the class definition of a class that will implement the StockWatch interface defined in Figure 3.

To create the stock service and make it available to clients, StockServer's main method must do several things: create and install a security manager, create a StockServer remote object, and make the StockServer object available to clients via a name facility (see Figure 4).

First, the main method creates a security manager to protect itself from code that may be downloaded during the RMI program execution. Some downloaded code cannot be trusted, so a SecurityManager provides a mechanism to "sandbox" the execution of downloaded code so that it cannot adversely affect the local machine. RMI provides an example security manager, RMISecurityManager, which implements security

policies similar to those provided by the security manager for applets running in a browser.[6]

Next, the main method creates the StockServer remote object and installs it in a name facility. In this example, the server creates its own registry and uses that as a simple name facility. RMI provides a name registry that may be shared by all servers running on a host, but any specific server may create and use its own registry if desired.

The call LocateRegistry.create Registry(2005) (see Figure 4) creates a registry on port 2005. The StockServer server object is bound in the registry via a call to Naming. rebind. The first argument in the rebind call is the URL (name) for the remote object. In the specified URL, the host defaults to the local host name, the port is specified as 2005, and the name is example.stock. StockServer. Note that the main method exits after these steps, but the server process remains alive as long as there are outstanding client references to the StockServer object. We ensure that the process remains alive by installing a reference to the remote object in the registry. (The system views the registry as a client, and so keeps the server process alive.)

```
public class StockApplet extends Applet
        implements StockNotify
{

    /**
    * Initialize applet: export the applet as remote object
    *that gets notified of stock updates.
    */
    public void init()
    {
        try {
            UnicastRemoteObject.exportObject(this);

            URL base = getDocumentBase();
            String serverName = "//" + base.getHost() + ":" +
                getParameter("registryPort") + "/example.stock.StockServer";
            stockWatch = (StockWatch)Naming.lookup(serverName);

            for (int i=0; i<name.length; i++) {
                stockWatch.watch(name[i], this);
                stockTable.put(name[i], new StockData(name[i], color[i]));
            }
        } catch (Exception e) {
            add(new Label("exception occurred during initialization; " + "check the log"));
            add(new Label(e.getClass().getName() + ": " + e.getMessage()));

            // fatal error
            System.out.println("got exception: " + e.getMessage());
            e.printStackTrace();
            return;
        }

        /* draw graph and labels... */
    }
```

Figure 6. Implementation of the applet init( ) method for an applet that makes a remote call to the stock server object defined in Figures 3 through 5.

The implementation of the server is not quite finished; we must still implement all its remote methods. Figure 5 shows an implementation of the watch method, which a client uses to request notification of stock price updates. The stock server adds the client (the StockNotify object) to its table (notifyTable) of remote objects to be notified of updates. A thread that sends periodic notifications to clients is started if the thread does not already exist.

We now turn to the implementation of an applet that acts as a remote object to receive stock updates. StockApplet extends the Applet class and implements the StockNotify remote interface so that it can receive stock updates via the update method. Since StockApplet extends the java.awt.Applet class, it cannot extend the java.rmi.server.UnicastRemoteObject RMI server class to automatically make the remote object available to accept incoming calls.

RMI provides another way to make a remote object available to the RMI runtime so that the object may receive calls from clients (in this case, the client would be StockServer). By making an explicit call to the static method Unicast RemoteObject.exportObject( ), an application or applet may export a specific remote object to the RMI runtime. Thus, in the applet's init method, the applet exports itself as a remote object to the RMI runtime.

```
/** Notification of stock updates for a particular time. */
public void update (Date date, Stock[] stock)
{
    System.out.println("StockApplet.update: " + date);
    // record date
    if (time.size() == MAX_UPDATES) {
        time.removeElementAt(0);
    }
    time.addElement(date);

    // record individual stock updates
    int numUpdates = time.size();
    for (int i=0; i<stock.length; i++) {
        StockData data =
            (StockData)stockTable.get(stock[i].symbol);
        if (data != null) {
            data.update(stock[i], numUpdates);
        }
    }
    repaint();
}
```

Figure 7. Implementation of the applet's remote method update( ), called by the stock server.

RMI also allows objects to be passed by value. Traditional distributed systems do not allow pass-by-value objects because such objects imply that the code is available for the object on the receiving end.

Next, the applet must register with the stock server to receive stock updates. This entails looking up the stock server by name in its registry, and then invoking the stock server's watch method for each stock. The name for the server is composed of the code base host (available via a call to base.getHost()), the registry's port (available via an applet parameter configured for this purpose), and the name of the stock server, example.stock.StockServer. The applet uses the Naming.lookup method to look up the StockWatch remote object by name, then it invokes the watch method for each stock for which it needs updates. See Figure 6.

In its role as a remote object, StockApplet implements the StockNotify interface, which consists of a single update method for receiving a stock price update. The implementation of this method simply stores the stock update information and repaints the graph to display the new information. See Figure 7.

For this example, we have elided much of the code that deals with the specifics of storing and displaying the stock information. For details, see the full code in the RMI documentation in the JDK 1.1 release.

This example illustrates how to write a simple peer-to-peer distributed application using RMI. It does not fully demonstrate all of RMI's features, but it does show that building an application in RMI is straightforward.

One of RMI's strengths is that an object may be passed polymorphically in an RMI call. That is, subtypes of the declared parameter types of a remote method can be passed in a remote method call, and the code for the subtype is downloaded as a side effect of the call.

Traditional distributed object systems do not allow parameter polymorphism because they are language-neutral and cannot support downloading code. For example, in the example in Figure 7, the server only cared that it was communicating with an object that implemented the StockNotify interface. However, the stub for the StockApplet that is passed to the StockServer supports the complete set of remote interfaces that the StockApplet supports. The StockServer could have discovered other remote interfaces that the StockNotify object implemented using standard Java language mechanisms (for example, instanceof).

RMI also allows objects to be passed by value. Traditional distributed systems do not allow pass-by-value objects because such objects imply that the code is available for the object on the receiving end. In multilanguage systems, code is not downloaded due to the system's heterogeneous nature. While it is theoretically possible to have a heterogeneous system that downloads compiled code, such a system would require code compilation for all possible platforms, for all possible classes, as well as a static set of classes.

We can give a powerful example of exploiting polymorphism in RMI by defining a compute server interface that executes the run method for arbitrary Task objects:

```
public interface Task implements
                                Serializable {
      Object run();
}
public interface ComputeServer extends
                           java.rmi.Remote {
      Object runTask(Task task) throws
                    java.rmi.RemoteException;
}
```

A client could define any object that implemented the nonremote Task interface and have its run method executed by a remote compute server. More specifically, it could send a task object to a ComputeServer in an RMI call; the task could be executed via a call to the task's run method, and the result could be returned to the caller. For example,

```
public class FFT implements Task {
      public FFT(args..) { .... }

      public Object run() {
             // computes FFT and returns
                                the result
      }
}
```

We could compute a fast Fourier transform (FFT) on a remote server simply by executing the following code:

```
ComputeServer comp =
(ComputeServer)Naming.Lookup
               ("//butterfly/computer");
FFT fft = new FFT(args...);
Object obj = comp.runTask(fft);
// display result returned in obj...
```

Such applications would be difficult if not impossible in traditional distributed object systems. Since RMI downloads code, this application is not only possible, but an easy extension of the general programming patterns used in the language.

IN JDK 1.1, RMI SUPPORTS a unicast reference type via the java.rmi.server.UnicastRemoteObject class. In the future, we will add several features to RMI, including the following:

- support for persistent remote references that are valid from run to run of a server—that is, such references persist and are valid over time;
- support for automatic server activation in conjunction with persistent remote references;
- support for RMI over secure transports such as SSL and/or SKIP;
- support for multicast remote references; and
- performance enhancements.

Beyond these enhancements to the basic RMI platform, we plan on using RMI's abilities to investigate other topics in the area of distributed systems. Since RMI allows systems to move code as well as data, and since it supports full polymorphic typing, we believe it is a base for investigations into such areas as agent technology and just-in-time software distribution.

References

1. A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, Vol. 2, 1984, pp. 39-59.
2. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley Longman, Reading, Mass., 1996.
3. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Longman, Reading, Mass., 1996.
4. G. Booch, *Object Oriented Design with Applications*, Benjamin/Cummings Publishing Company, Inc., Redwood City, Calif., 1991.
5. A.D. Birrell et al., "Network Objects," Technical Report 115, Digital Equipment Corporation Systems Research Center, Palo Alto, Calif., 1994.
6. A. van Hoff, S. Shaio, and O. Starbuck, *Hooked on Java*, Addison-Wesley Longman, 1996.

Ann Wollrath is a senior staff engineer with JavaSoft, where she is the lead designer and project lead of the Java Remote Method Invocation system. Prior to joining JavaSoft, she researched reliable, large-scale distributed systems in Sun Microsystems Laboratories and parallel computation at MITRE Corporation.

Availability

The Java Remote Method Invocation System is part of the basic Java Development Kit release 1.1. Versions of JDK 1.1 for Solaris, Windows95, and WindowsNT can be obtained via the Web at http://www.javasoft.com/.

Wollrath received an MS from the University of Massachusetts, Lowell, and a BS from Merrimack College. She is a member of the IEEE and the ACM.

Jim Waldo is a senior staff engineer with JavaSoft, where he is responsible for the overall architecture of pure Java distributed systems. Prior to joining JavaSoft, he was principal investigator for the large-scale distribution project in Sun Microsystems Laboratories.

Waldo received a PhD from the University of Massachusetts, Amherst. He also holds master's degrees in philosophy and linguistics and a BA from the University of Utah. He is a member of the IEEE and the ACM.

Roger Riggs is a senior staff engineer with JavaSoft, where he is responsible for the object serialization system. Prior to joining JavaSoft, he did research in multimedia and large-scale distributed computing at Sun Microsystems Laboratories. Riggs received a BS from Carnegie Mellon University. He is a member of the ACM.

Direct questions concerning this article to Ann Wollrath, Sun Microsystems, Inc., Mailstop UCHL03-304, 2 Elizabeth Dr., Chelmsford, MA 01824-4195; ann.wollrath@sun.com.